AD-A062 922 NAVAL RESEARCH LAB WASHINGTON D C F/G 9/2
EVALUATING SOFTWARE DEVELOPMENT BY ERROR ANALYSIS. THE DATA FRO--ETC(U)
DEC 78 D M WEISS

UNCLASSIFIED NRL-8268 NL

1 OF 1

AD
A062922

END
DATE
FILMED
3--79
DDC

13

NRL Report 8268

Evaluating Software Development by Error Analysis: The Data From the Architecture Research Facility

David M. Weiss

Information Systems Staff
Communications Sciences Division

December 22, 1978

NAVAL RESEARCH LABORATORY
Washington, D.C.

Approved for public release; distribution unlimited.

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
NRL Report 8268

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
(16) EVALUATING SOFTWARE DEVELOPMENT BY ERROR ANALYSIS, THE DATA FROM THE ARCHITECTURE RESEARCH FACILITY.

5. TYPE OF REPORT & PERIOD COVERED
(9) Interim rep~~ort~~, on one phase of a ~~continuing NRL~~ Problem

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
(10) David M. Weiss

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Naval Research Laboratory
Washington, D.C. 20375

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
NRL Problem 54B02-18
62721N XF21-241-021

11. CONTROLLING OFFICE NAME AND ADDRESS
Naval Electronic Systems Command
Washington, D.C. 20360

12. REPORT DATE
December 22, 1978

13. NUMBER OF PAGES
23

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)
(11) 22 Dec 78 (12) 24 p.

15. SECURITY CLASS. (of this report)
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
(14) NRL-8268
Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)
(16) F21241 (17) XF21241021

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Architecture Research Facility	Software errors
Computer programming	Software error analysis
Software	
Software engineering	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

In software engineering, it is easy to propose techniques for improving software development but difficult to test the claims made for such techniques. This report suggests an error analysis technique for use in gathering data concerning the effectiveness of different software development methodologies. The principal features of the error analysis technique described are formulating questions of interest and a data classification scheme before collection begins, and interviewing of system developers concomitant with the development process to verify the accuracy of the data. The data obtained by using this technique during the development of a medium-size software development

(Continued)

over

251 950

79 01 03 012

20. ABSTRACT (Continued)

project are presented. This project was known as the Architecture Research Facility (ARF) and took about 10 months and 192 man-weeks of effort to develop. The ARF designers used the information-hiding principle to modularize the system, and interface specifications and high-level language coding specifications to express the design. Several error-detection aids were designed into the system to help detect run-time errors. In addition, quality control rules were established that required specification review before coding, and code review after compilation but prior to testing. A total of 143 errors was reported. Analysis of these errors showed that there were few problems caused by intermodule interfaces, that error corrections rarely required knowledge of more than one module, that most errors took less than a few hours to fix, and that error-detection aids detected more than half the errors that were potentially detectable by them.

CONTENTS

INTRODUCTION 1

THE ARF PROJECT 2

Design Goals 2
Approach to Achieving Goals 3
Development Organization 4

ERROR ANALYSIS: ANSWERING QUESTIONS OF INTEREST 6

Users of Error Data 8

ARF ERROR DATA 8

Error Classification 8
Reporting Requirements 9
Accuracy Checks 9

ARF ERROR ANALYSIS 9

Errors Related to Modularization 10
Methods of Error Detection 12
Error-Correction Effort and Methods 13
Limitations and Problems 15

CONCLUSIONS 17

ACKNOWLEDGMENTS 17

REFERENCES 17

APPENDIX – Modules of the ARF 19

EVALUATING SOFTWARE DEVELOPMENT BY ERROR ANALYSIS: THE DATA FROM THE ARCHITECTURE RESEARCH FACILITY

INTRODUCTION

In science, one usually proposes a hypothesis, performs an experiment to gather data, then uses the data to verify or discredit the hypothesis. Unfortunately, in software engineering it is remarkably easy to propose hypotheses, and remarkably difficult to test them. Accordingly, it is useful to seek methods for testing software engineering hypotheses. This report describes one such technique and presents the data obtained from its use in a medium-size software development project. We restrict our attention to the area of software development methodologies, and show how one may gather data concerning the effectiveness of different methodologies in producing correct software. The technique discussed here is based on collecting data concerning errors. The kind of data to be gathered is established at the start of development, and data gathering proceeds in parallel with development. The kind of data needed and the analysis performed on the data depend on the claims made for the methodology being used.

Later sections of this report describe in greater detail the steps to be taken to evaluate a methodology by error analysis. It should already be evident, however, that a true experimental evaluation will not result. There is no control group, possible confounding factors are neither neutralized nor eliminated, and statistical hypothesis testing is not likely to be useful. Nonetheless, there is still much to be gained from the kind of evaluation we suggest. One can discover limitations and advantages of various techniques with respect to error prevention, detection, and repair. An additional benefit is that one can gain a great deal of insight into the software development process. The kinds of activities going on at different stages of the development cycle become apparent. The more troublesome or time-consuming an activity is, the more visible it becomes.

The error analysis technique described here was tested on an in-house software development project at the Naval Research Laboratory. The purpose of the project was to develop a facility for simulating different computer architectures, to be known as the Architecture Research Facility (ARF). The function of the ARF is described in the next section.

Besides producing a working simulator, the ARF designers had specific goals for their design. They were careful to select design techniques that they felt would help to achieve those goals. The first section contains a discussion of both the design goals and the design techniques. The error data that were collected permitted the designers to have some measures of the effectiveness of the techniques they used. The approach to obtaining the error data is discussed in the second section, and the third section is a discussion of the results of the data collection and analysis. The shortcomings of error analysis and the problems encountered in collecting and analyzing data are described in the fourth section. Finally, some conclusions that can be drawn about the ARF project and the usefulness of error analysis in evaluating software

Manuscript received July 21, 1978.

development methodologies are presented in the fifth section. Readers interested only in a discussion of the techniques involved in data collection and analysis should read the second, third, and fifth sections. Those interested in the ARF data should read the first and fourth sections.

The author was not involved at any time in the actual development of the ARF. He was only involved in the collection and analysis of the error data.

THE ARF PROJECT

The purpose of the ARF project was to develop a facility for simulating different computer architectures based on a description of the target architecture written in the Instruction Set Processor (ISP) language [1]. Different computer architectures may be evaluated by running programs on the simulated computers. The ARF enables the user to monitor the simulation interactively, allowing him to observe, at his option, the state of selected components of the target computer. As an example, a user wishing to simulate a general register machine might want to run a matrix-inversion program, observing the number of memory fetches, additions, multiplications, and total instructions executed.

A complete description of the structure of the ARF simulator is available elsewhere [2], and is not needed here. Briefly, to simulate a machine, the ARF uses a set of tables used to describe the machine being simulated and its state, a module to perform instruction simulation, and a module to handle the interface to the user. The machine description contained in the tables is produced by an ISP compiler (written at Carnegie-Mellon University).

The ARF was developed by a team of nine people (excluding consulting and secretarial support), eight of whom did not participate full time over the entire development phase. There were five principal designers, one of whom doubled as project manager and some of whom also coded, and two principal coders. The remaining two people did some minor design and coding, and one provided a test and debug support package. Secretarial support ranged from one to four people, as needed. During the early design period, there was one consultant who strongly influenced several major design decisions. The ARF took about ten months and 192 manweeks, exclusive of consulting and secretarial support, to develop. During this time, various designers and coders joined and left the project.

Design Goals

The primary goal of the ARF designers was to produce a working simulator that would permit the simulation of small target-machine programs. The simulator was expected to be fast enough to run in an interactive mode, so that the user could see results of the simulation without having to wait for completion of a batch job. By the time the design was completed, the following additional goals had been added.

- Rather than developing the whole system at one time, the ARF was to be done using the family approach to software development [3]. The system would be built in three main stages. Each stage would produce a member of the ARF "family" of programs, providing different facilities.

- The information-hiding principle [4] was to be applied to conceal design decisions that were expected to change during the lifetime of the ARF. As an example, early in the design stage, it was expected that two different versions of the ARF would be developed: initially, one to run on a PDP-10, and later one to run on a PDP-11. Table sizes and structures were expected to be markedly different between the two versions. Information hiding was expected to ameliorate the problems in producing two consistent versions. Although planned, the move to the PDP-11 was never attempted. As a result, the effectiveness of the design in aiding transportability cannot be assessed.

- At the possible expense of some run time performance, several debugging aids were designed into the system to make development easier. These included

a. A method for detecting errors involving improper access to table entries,

b. A consistent execution-time error reporting scheme for table interface functions that preserved the name of the routine in which the error occurred and reported a code associated with the error, and

c. A mechanism for inserting, and turning on and off, debugging code through the use of a compile-time preprocessor.

The family approach is mentioned as an example of a technique that could not be evaluated by error analysis as described here. As a result, further elaboration of the ARF family will be omitted. The following sections contain a more complete description of the application of information hiding and the ARF debugging aids.

Approach to Achieving Goals

Understanding of parts of the error analysis requires some knowledge of how the ARF was designed. A brief description of the approach taken to achieving the design goals is given here.

Modularization Considerations

The ARF was modularized in a conscious effort to hide certain design decisions. Some decisions hidden were the structure of the tables used to store target-machine operations and the state of the target-machine (hereafter referred to as the descriptor tables), the representation used by the ISP compiler for the object code it produced, and the implementation of the user interface. (The appendix contains a short description of the eight ARF modules and the design decisions hidden in each.) One example is the table access module, which was designed to conceal the implementation of the descriptor tables. Access to table entries was provided by a set of table interface functions contained in the table access module. These functions were the only ARF routines containing references to the variables used to implement the tables. Routines in other modules needing access to table entries had to use the table interface functions for such access.

Binding Mechanism and Table Interface Functions

In the interests of decreasing debugging time and difficulty, a special mechanism was designed and implemented to detect some types of improper access to descriptor table entries. This mechanism, known as the binding mechanism, was used by routines not belonging to the table access module. The mechanism consisted of an implementation of a set of rules defining the legality of referencing table entries. No more than one entry in each table could be referenced at any time. The entry to be referenced had to be declared before any reference occurred. The operation of declaring an entry to be referenced was known as binding (freeing the entry was unbinding), and the declared entry was known as the bound entry. Binding and unbinding were accomplished by calling special table interface routines designed for the purpose. An error was generated any time access was attempted to a table entry that was not bound.

Error-Reporting Mechanism

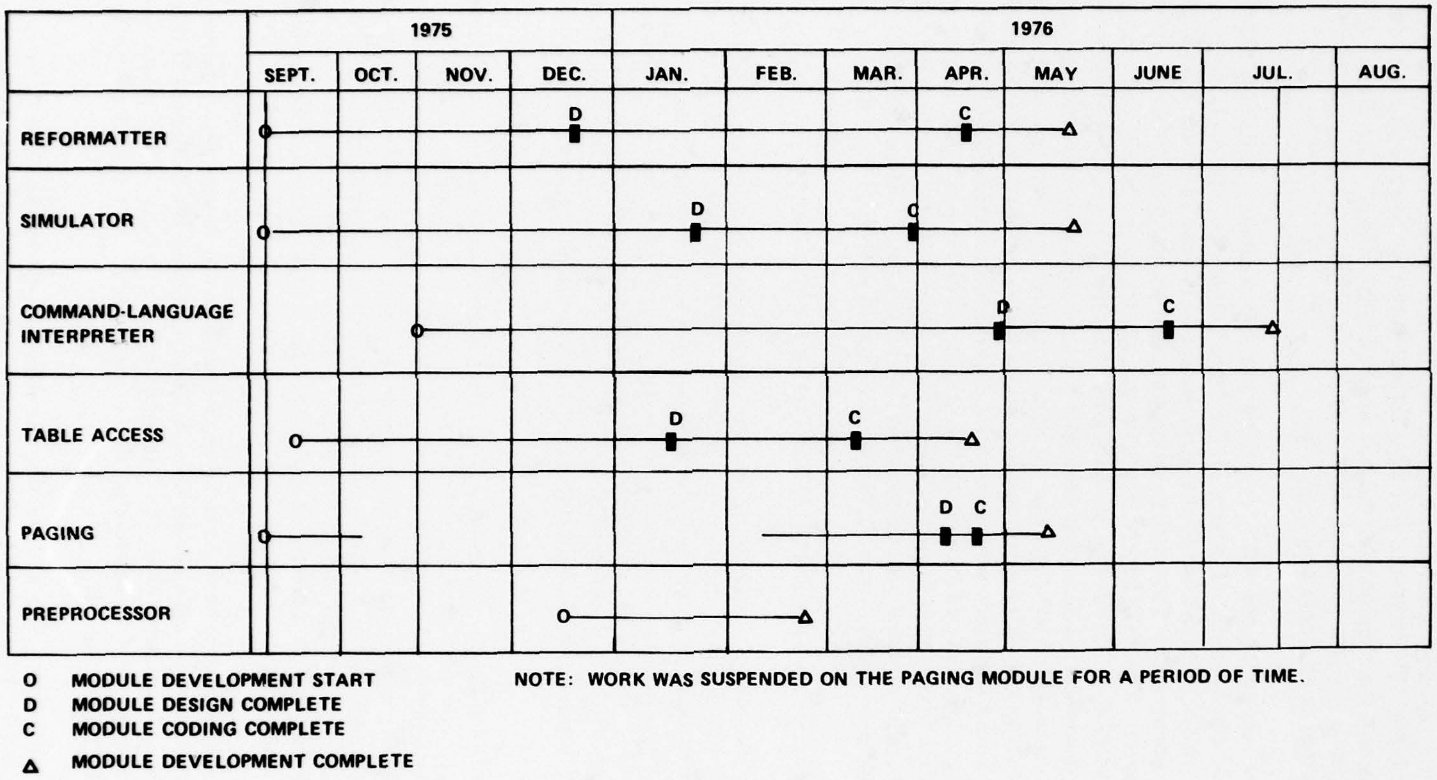
Improper binding errors were only one class of errors expected to occur that involved table access. Because there were several different tables and many different table entry types, a consistent scheme for reporting errors involving table interface functions was used. Each table interface function returned a parameter containing an error code. The value of the parameter denoted the function in which the error was discovered and identified the type of error. Function identifiers and standard error types were contained in a parameter file that was accessed at compile time through the use of the preprocessor. The error reporting mechanism was explicitly designed to facilitate debugging and testing of the ARF. All ARF programs were written so that the error reporting mechanism could be turned off without affecting the results of using the ARF. When the mechanism was turned off, simulation speed increased by about a factor of 2.

Preprocessor

The ARF programs were written in ANSI standard FORTRAN on a PDP-10. Because the standard lacked some features wanted by the ARF developers, a preprocessor was written to extend the language's capabilities. The three main capabilities provided by the preprocessor were compile-time inclusion into programs of previously defined files, special identification of debug code (the programmer could use the preprocessor to convert all code identified as debug into either comments or standard FORTRAN statements), and substitution of constants for identifiers (permitting the use of names for compile-time constants).

Development Organization

Figure 1 is a calendar of development time for each module in terms of design, coding, and testing. Once the basic modular structure of the ARF was established, the usual procedure followed for each module was to produce and review a series of design documents. The initial document(s) named and informally described the functions and data structures that composed the module, and defined the module's interface to its users. After this basic design was reviewed and found acceptable by all other designers, high-level language coding specifications were written for each function of the module. Each coding specification was reviewed by a designer other than its author prior to being released for coding. Typically, two to four versions of the design and interface specifications were produced before the coding specification was written, and two to four versions of the coding specifications were produced before coding started. FORTRAN code was written from the coding specifications, compiled, and then reviewed by someone other than the coder. After the code review, the coder debugged the routines he wrote, and delivered them for testing. A tester, usually other than the coder or designer, was then selected. As development proceeded and design changes were made, the



NOTE: WORK WAS SUSPENDED ON THE PAGING MODULE FOR A PERIOD OF TIME.

Fig. 1 — ARF development calendar

design and coding specifications were updated. Eight or nine versions of some documents were eventually produced. These procedures were followed for the major modules of the ARF. The utility routines and the preprocessor did not undergo this process. The preprocessor was written in SNOBOL as a single program, and some of the utilities, such as the byte-handling routines, were written in assembly language.

The final version of the ARF, excluding the ISP compiler, contained 253 routines and 21,831 lines of source code, of which 11,796 were comments. Most of these routines were coded in FORTRAN, with the aid of the preprocessor to include common blocks, global parameter files, and debug code. The coding, debugging, and testing were all done interactively on a PDP-10 computer. The FORTRAN language was used because it was considered most transportable to the PDP-11 (recall that an early goal of the project was to produce a version of the ARF for both machines). The coding standards [5] used for the project established ANSI FORTRAN as the standard ARF language. Coding specifications were all written in high-level languages, with the choice of language for any particular module left to the author of the specification. Three languages were chosen for this purpose: SIMPL-T [6], BLISS [7], and an ALGOL-like language devised by one of the designers for his use. The project manager had exclusive, on-line access to a library of source and object code for tested ARF routines. The documentation, also on-line, was kept current with the library. More details of the development methodology and project management can be found in Ref. 2.

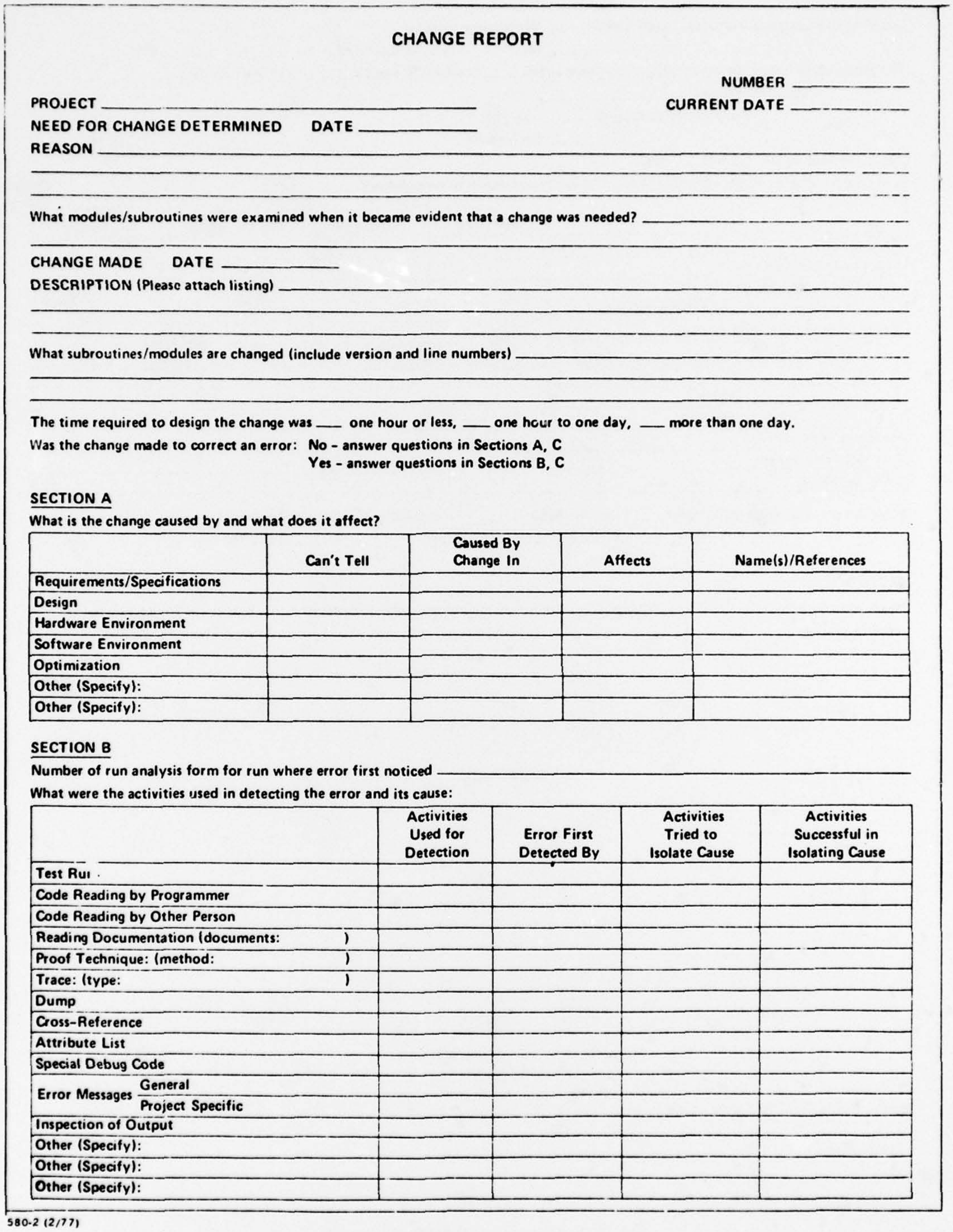
ERROR ANALYSIS: ANSWERING QUESTIONS OF INTEREST

Much of the effort involved in applying error analysis comes before and during the development of the software under study. Before development begins, the questions of interest to the study must be formulated, a data classification scheme must be selected, a questionnaire for data collection must be devised, and the subjects involved should be trained in filling out the questionnaires. During the development stage, the error analyst(s) must review each questionnaire for consistency, completeness, and accuracy. If there is doubt about the appropriate classification of errors, or if data are incomplete or inconsistent, the developers involved must be interviewed.

A number of different suggestions for selecting error classification schemes exist in the literature [8-12]. The scheme chosen should be based on the questions of interest. Postponing definition of the scheme until most data have been collected may cause a number of interesting questions to be left unanswered. As an example, if one is concerned with the utility of code reading in producing reliable software, pertinent data to be collected for each error consists of the method used for the detection of that error. If this information is not collected during the development process, it is unlikely that one will be able to discover the number of errors detected by code reading.

Once a classification scheme is defined, it is possible to devise a questionnaire to collect data of interest. Questionnaire design is an art that has been much discussed elsewhere [13] and will not be treated here. We only mention that the original ARF questionnaire design, driven by considerations of completeness, was 11 pages long. The version in use by the end of the project was 2 pages (Fig. 2).

As part of collecting data and training the people involved, procedures for collecting data must be established. The definitions of error and error correction, the kinds of errors to be



CHANGE REPORT

NUMBER ____________

PROJECT ________________________ CURRENT DATE ____________

NEED FOR CHANGE DETERMINED DATE ____________

REASON ____

____

____

What modules/subroutines were examined when it became evident that a change was needed? ____________

____

CHANGE MADE DATE ____________

DESCRIPTION (Please attach listing) ____________________________

____

____

What subroutines/modules are changed (include version and line numbers) ____________

____

____

The time required to design the change was ___ one hour or less, ___ one hour to one day, ___ more than one day.

Was the change made to correct an error: No - answer questions in Sections A, C
Yes - answer questions in Sections B, C

SECTION A

What is the change caused by and what does it affect?

	Can't Tell	Caused By Change In	Affects	Name(s)/References
Requirements/Specifications				
Design				
Hardware Environment				
Software Environment				
Optimization				
Other (Specify):				
Other (Specify):				

SECTION B

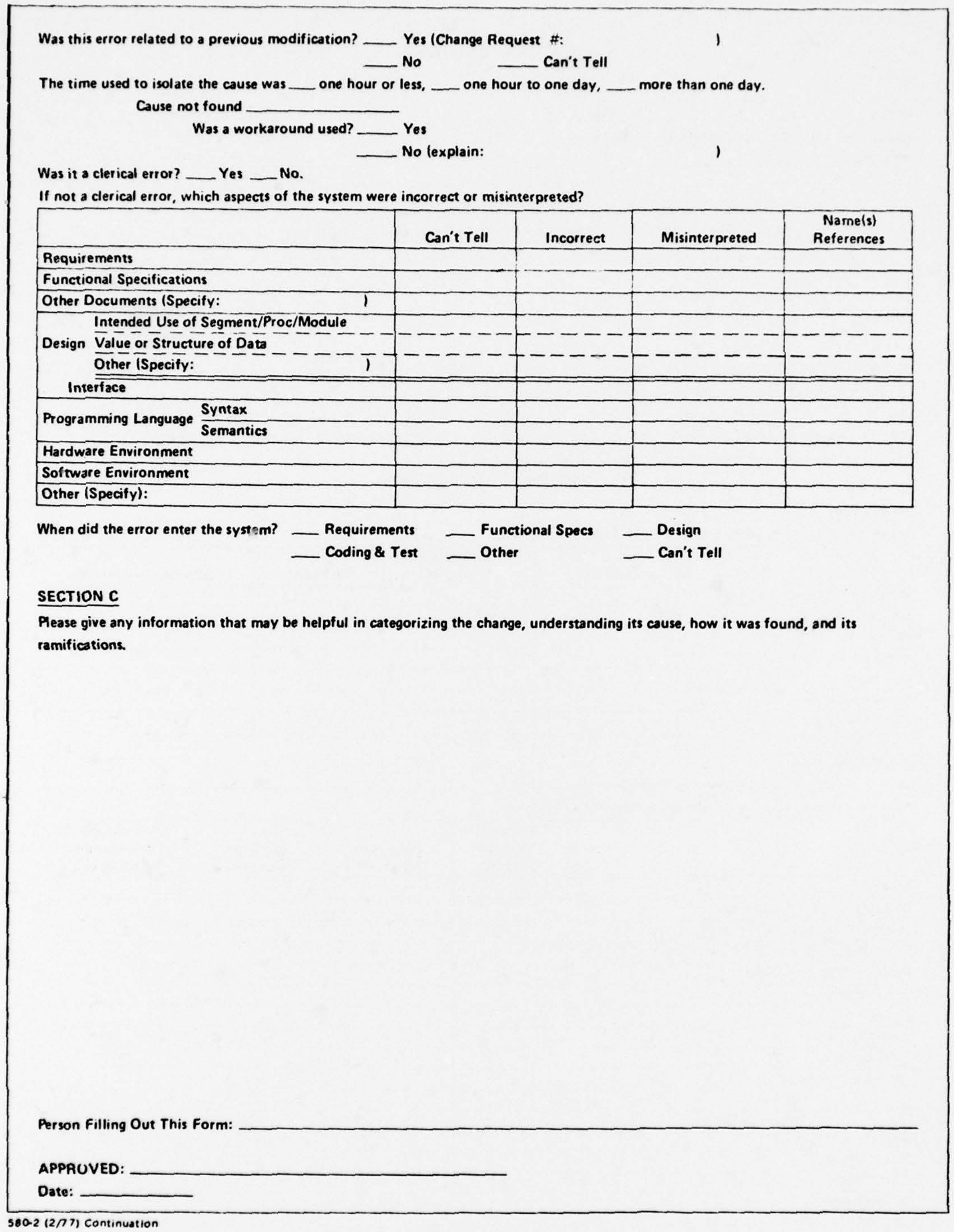
Number of run analysis form for run where error first noticed ____________

What were the activities used in detecting the error and its cause:

	Activities Used for Detection	Error First Detected By	Activities Tried to Isolate Cause	Activities Successful in Isolating Cause
Test Rui				
Code Reading by Programmer				
Code Reading by Other Person				
Reading Documentation (documents:)				
Proof Technique: (method:)				
Trace: (type:)				
Dump				
Cross-Reference				
Attribute List				
Special Debug Code				
Error Messages General				
Project Specific				
Inspection of Output				
Other (Specify):				
Other (Specify):				
Other (Specify):				

580-2 (2/77)

Fig. 2 — Change Report Form



Was this error related to a previous modification? _____ Yes (Change Request #:)
_____ No _____ Can't Tell

The time used to isolate the cause was ____ one hour or less, ____ one hour to one day, ____ more than one day.

Cause not found ____________________

Was a workaround used? _____ Yes
_____ No (explain:)

Was it a clerical error? ____ Yes ____ No.

If not a clerical error, which aspects of the system were incorrect or misinterpreted?

		Can't Tell	Incorrect	Misinterpreted	Name(s) References
Requirements					
Functional Specifications					
Other Documents (Specify:)					
Design	Intended Use of Segment/Proc/Module				
	Value or Structure of Data				
	Other (Specify:)				
Interface					
Programming Language	Syntax				
	Semantics				
Hardware Environment					
Software Environment					
Other (Specify):					

When did the error enter the system? ____ Requirements ____ Functional Specs ____ Design
____ Coding & Test ____ Other ____ Can't Tell

SECTION C

Please give any information that may be helpful in categorizing the change, understanding its cause, how it was found, and its ramifications.

Person Filling Out This Form: ____

APPROVED: ______________________________

Date: __________

580-2 (2/77) Continuation

Fig. 2 (Continued) — Change Report Form

reported, and the times at which errors are to be reported must be made clear to all involved. In large development projects there are usually formal procedures for updating program libraries. These procedures can be used by error analysts to account for all changes, and hence all corrections, to programs already in the library.

Users of Error Data

The analyses discussed in the ARF Error Data section show some kinds of information that can be derived from historical error data. They were selected as examples because they are of interest to a variety of people involved in software development. As examples, a project manager might be interested in the overall success of the development methodology and the phase(s) of the development cycle that are most troublesome; a system designer might be interested in whether or not he achieved his design goals; a software engineer might be interested in the effectiveness of the design methodology and in the techniques found useful for finding and correcting errors; a programmer might be interested in the most effective error-detection techniques; and an error analyst might be interested in knowing major sources of errors, the success of techniques specifically designed to detect and correct errors, and error rates. Other analyses that were or could be performed on the ARF error data have not been reported here because results were either inconclusive or unreliable. These include calculating the fraction of errors that occurred in each ARF module, the fraction of errors detected by various techniques before testing or coding began, and the number of errors detected at design time that would have been detected by the error-detection and -reporting mechanism.

ARF ERROR DATA

Error Classification

The classification scheme used in collecting and analyzing ARF error data was defined at the start of ARF development and refined somewhat during the design phase. The scheme was defined so that information concerning the methodology and the success or failure of the design would be available for analysis after collection of the data. The issues of interest were the usefulness of interface and coding specifications, of code reading for error detection, and of the built-in error-detection mechanisms. The success in achieving design goals was assumed to be related to the number of errors involving module interfaces (related to the use of information-hiding and interface specifications), the fraction of errors involving specifications, the fraction of errors detected by code reading, the fraction of errors detected and reported by the error-reporting mechanism, the fraction of errors associated with improper binding, and the difficulty of correcting errors. Also of interest were the number of attempted corrections for each error, and the misunderstandings involved in different errors. More detailed discussions of these parameters will be presented later in this section.

The ARF was not sufficiently big to warrant the use of formal change procedures, and designers and programmers were relied on to report their own errors. The only available checks on this process were revisions to the interface and coding specifications, which were published as needed, and the awareness of project management of the day-to-day progress of the development effort. Since the project manager and her supervisor were both involved in the design and specification of the ARF, and approved all design changes, their support and example in reporting errors were invaluable (the error analysis project received strong support from ARF project management during the entire development period). Newly hired coders were informed when they started work that error reporting was considered part of their duties. Furthermore, error data were never used in assessing programmer efficiency or competence, and no attempt to allocate blame for committing errors was ever made.

Reporting Requirements

Error reporting for the purposes of error analysis was required on all modules except the ISP compiler, which was not developed at NRL. An error was considered to be a discrepancy between a specification and its implementation. Specifications included requirements, design, interface and coding specifications, coding standards, and program documentation. All errors were expected to be reported when the error was corrected (no cases occurred where there was a delay of more than about a week between error detection and error correction). For specification and design errors prior to coding this was when a correction to the appropriate document was ready to be submitted. For errors requiring code changes this was when a change to the library was ready. The only errors not required to be reported were certain classes of clerical errors, such as misspellings resulting from keypunch errors and simple syntactic errors such as mismatched parentheses and omitted commas. Although these errors were not required to be reported, coders were encouraged to report them (one result of this policy was that there were more errors in the clerical category than any other).

Accuracy Checks

Each error report was examined by an ARF error analyst after submittal. If there was any question concerning the circumstances of the error and its detection, or the categorization of the error, the person who completed the error report form was interviewed by the error analyst. This policy was an attempt to ensure consistent categorization of errors. In addition, at the end of the ARF development, an ARF programmer reviewed all the error reports for accuracy and consistency of categorization.

ARF ERROR ANALYSIS

Although there is no good way to measure the success of the error reporting procedures, we believe that nearly all errors found at the coding specification stage or beyond were reported. We have less confidence that all design errors found prior to writing coding specifications were reported. Careful monitoring of errors did not start until December 1975. Some design errors were found and corrected in design review meetings held prior to December for which no written records exist. Furthermore, it was sometimes difficult to distinguish between design improvements and design errors. The first error was reported in early January 1976, and was an error in translating coding specifications to code. The last error was reported in April 1977, and was a data type error originating in a coding specification.

A total of 143 errors was reported during the development and use of the ARF through July, 1977 (one year after the end of development). The data collection scheme used allows the errors to be categorized according to methods of detection, sources of misunderstanding, module of occurrence, relationship to previous modifications, difficulty of correction, and size of erroneous subroutine (if applicable). The following sections discuss the different categorizations with regard to the questions of interest described in the section on error data.

Errors Related to Modularization

The effectiveness of a particular modularization strategy is difficult to measure directly. Based on the claims for the information hiding approach, it was assumed that the goodness of the ARF's modular structure was correlated with the following measurable factors:

- The percentage of errors resulting from interface misunderstandings

- The effort involved in fixing interface misunderstandings

- The percentage of erroneous changes

- The percentage of errors that required an understanding of several ARF modules to correct or that required corrections to be made in several modules.

Table 1 is a categorization of ARF errors according to the misunderstanding that was the source of the error. (For completeness, clerical errors and careless omissions, which are not misunderstandings, are also shown.) An error is classified as a misunderstanding if it is a result of an incorrect assumption. As an example, one error resulted from a programmer's assuming that the ARF user input routines removed delimiters from character strings containing user commands. This assumption was false, and resulted in one type of user command being rejected as invalid. About half (54%) of ARF errors were the result of a misunderstanding of requirements, design, interface, coding specifications, a language (including FORTRAN, the ARF preprocessor, a specification language used in writing coding specifications, or the text editor used to enter source code), or of the ARF FORTRAN coding standards. Most of the remainder (39%) were clerical errors, and there were some (10%) careless omissions that could not be classified as misunderstandings. (Typical careless omissions were omitted declarations of variables, common blocks, or files, either in the source code or in a coding specification. In these cases the coder or designer realized that the declaration was needed but forgot to include it when writing the code or specification.) The interface between modules consists of the assumptions made by the modules about each other. An interface error is then an incorrect assumption about one module by another. Examples of ARF interface errors are subroutines of one module called by subroutines of another module in circumstances that violated the former's specifications (e.g., improper reinitialization call or incorrect choice of data conversion routine), use of incorrect calling sequences between subroutines of different modules, and unexpected argument values returned from a subroutine.

Table 1 — Misunderstandings as Sources of Errors

Category	Number of Errors	Percent of Total Errors
Requirements	8	6
Design (excluding interfaces)	27	19
Interface	9	6
Coding specifications	18	13
Language	12	8
Careless omission	14	10
Coding standards	3	2
Clerical	52	36

Two areas often considered to be the most troublesome, requirements and interfaces, were among the smallest sources of misunderstandings. These two categories together comprised 12% of the total errors. Furthermore, when the effort involved in fixing (finding the cause of and correcting) an error is considered (see Table 2), all but one of the interface misunderstanding errors were rated easy (i.e. took less than a few hours) to fix. Misunderstandings of requirements, although small in number, were more difficult to fix than other error types. (Difficulty of error correction is analyzed in more detail in a later section.) Most of these errors involved a misunderstanding of the semantics used by the ISP compiler.

Table 2 — Effort Involved in Fixing Errors

Category of Misunderstanding	Effort to Fix		
	Easy (less than a few hours)	Medium (a few hours to a few days)	Hard (more than a few days)
Requirements	4	3	1
Design (excluding interface)	19	8	0
Interface	8	1	0
Coding specifications	14	4	0
Language	9	3	0
Careless omission	13	1	0
Coding standards	3	0	0
Clerical	50	2	0

Additional analysis of the ARF errors, not reflected in the tables, shows that there were eight errors (6% of the total) for which determining the cause required understanding of more than one module. By "understanding" we mean that the error corrector had to know some of the details of how the modules operated. As an example, one assumption made in the design of the simulator module was that the minus sign always denoted a binary operator. This assumption was incorrect for some cases and resulted in an error. To determine the cause of the error it was necessary to know some of the details of how code was generated by the ISP compiler. In each of the 8 cases, designing and implementing the correction required understanding of only one module.

Unfortunately, aside from error corrections, no record of modifications made to the ARF was kept, so no complete measure of the number of errors committed in making changes is available. Examination of the errors does show that only one error was the result of an error correction, and that 11 errors (8% of the total) resulted from a change made for different reasons.

Four measures related to modularization have been calculated in the foregoing: errors resulting from interface misunderstandings, effort involved in fixing interface misunderstandings, errors whose cause determination required an understanding of more than one module, and errors committed in making error corrections. The values of these measures tend to indicate that information needed to modify routines was local to modules, and that interfaces between modules were easy to understand.

Methods of Error Detection

The ARF developers paid much attention to issues involving error detection. The rules established for writing and reviewing design, interface, and coding specifications and code were designed to find errors as early as possible. The table access module, which contained the most complex data structure and which was expected to change most during the lifetime of the ARF, had special error-detection and -reporting mechanisms built into it. Table 3 is a summary of errors according to the method of discovery. As previously noted, we believe that there were unreported design and specification errors that occurred early in the project's development phase. As a result, the number of errors detected by reading specifications or code (29%) must be considered a lower bound.

Table 3 — Methods of Error Detection

Category	Number of Errors	Percent of Total Errors
Detected from program execution	58	40
Detected by reading code or specifications	41	29
Other (FORTRAN compiler, preprocessor, etc.)	44	31

Review of Specifications and Code

Errors detected by reading specifications or code can be further categorized as shown in Table 4. As shown, most of these errors were detected by "quality control" inspections. Recall that the "quality control" rules for the ARF required that all coding specifications and code undergo a quality control inspection. Since all ARF errors must be viewed as potentially detectable by these means, in terms of numbers of errors the quality control rules do not appear especially effective, detecting only 17% of all errors (all of which were easy to fix). The more traditional method of running the program to find errors appears to have been the most effective, since 40% of all errors were detected as a result of program execution. It is important to note, however, that all but one of the errors discovered by reading code or specifications were easy to fix, whereas 21 (36%) of the errors detected by program execution were medium or hard to fix. The usefulness of the quality control rules is predicated on early detection and correction of errors. Evaluation of that usefulness depends on knowing the cost of *not* using the rules. The evaluation could be done by assuming that quality control inspections were not used and determining the cost of fixing the 24 errors that were found by such inspections. This requires knowing (or estimating) when those errors would have been detected and how much effort then would have been involved in fixing them. We did not feel it was possible to estimate that effort accurately.

Error Detection Mechanism

Evaluating the error detection and binding mechanisms requires examining just those errors that were potentially detectable by those mechanisms. As shown in Table 5, 43% of the errors detected by executing the program involved access to the descriptor tables. More than

Table 4 — Errors Detected by Reading Specifications or Code

Category	Number of Errors	Percent of Total Errors
The original programmer in "considering his program"	7	5
A "quality control" inspection by someone other than the original programmer	24	17
Inspection by someone other than the original programmer for some purpose other than quality control	7	5
During translation from specifications to code	3	2

Table 5 — Errors Detected by Error-Handling (and Binding) Mechanism

Category	Number of Errors	Percent of Total Errors
Not involved in table access	33	23
Involved in table access but *not* detected by error handler	11	7
Involved in table access and detected by error handler*	14	10

*Of these, 13 were detected by the binding mechanism.

half (56%) of these were reported by the error handling mechanism. All but one of the errors reported in this way were detected by the binding mechanism. On the other hand, two errors detected by improper binding were created by the binding mechanism itself, i.e. they were cases where the attempted table access was proper, but the appropriate binding operation had been omitted. Finally, of the 58 errors detected at execution time, 24% were reported by the error handler (22% by the binding mechanism part of the error handler). Based on this analysis, the built-in error detection mechanisms seemed quite effective in detecting table access errors, which comprised slightly less than half of the errors detected by running the program.

In summary, the run-time error-detection mechanism detected more than half of the table access errors. Because the cost savings associated with the quality control rules cannot be accurately estimated, it is not possible to judge their utility, but only note that they were responsible for early detection of 17% of the errors, all of which were easy to fix.

Error-Correction Effort and Methods

The amount of effort that should be and is expended in testing is a topic of considerable debate in the literature [14,15]. An important factor in estimating test effort is the difficulty of correcting errors. This factor might also be used as a qualitative indicator of the success of the project's development methodology. For each ARF error reported, an estimate of the difficulty of fixing (i.e., finding the cause of and correcting) the error was requested. The categories used were easy (took less than a few hours), medium (a few hours to a few days), or hard (more than a few days). Table 6 gives the distribution of errors according to difficulty to fix. Only one error required more than a few days to fix. The easy-to-fix errors constituted 84% of the total number. Of the medium-rated errors, at least half took a day or less to correct (there is insufficient information available to determine whether or not the other half took more than one day). It has been noted that the later in the development cycle an error is found, the harder it is to correct. In an attempt to verify this for the ARF, the difficulty of correction of errors was compared with the dates on which they were first observed. All but one of the errors rated medium and hard found during development were discovered in the last month (last 10%) of development. Most of this time was spent in integration testing. (The one medium-rated error found earlier was related to a misuse of the text editor used for entering source code, and had nothing to do with the design or implementation of the ARF.) During this period, nine errors rated easy were also found. Unfortunately, one cannot conclude from this that it was the late discovery of these errors that made them difficult to correct, but can only note the correlation. Further analysis into the nature of the difficult-to-fix errors is necessary before conclusions as to cause may be drawn.

Table 6 — Difficulty of Fixing Errors

Category	Number of Errors	Percent of Total Errors
Easy (less than a few hours)	120	84
Medium (a few hours to a few days)	22	15
Hard (more than a few days)	1	1

In addition to difficulty of correction, methods of error correction were also analyzed. For each error, the method of determining both the reason and required fix for the error were requested. Any combination of the following four categories was allowed: study of the algorithm in the documentation, inspection of code, running test cases, and other. This request was answered for all but 27 of the errors (24 of these 27 were spelling errors in identifiers probably caught by a compile-time diagnostic). Table 7 shows the results of this categorization. After inspection of code, study of the documentation (other than comments in the code) was the most popular technique. This reflects the effort made to keep the documentation current and useful. Some responses in the "other" category were executing code by hand, compilation output, discussion with ISP compiler designer, use of special debugging code, and dump reading. It is significant to note that only once was a dump needed, and only once was the debugging code inserted for the express purpose of finding a particular bug.

In summary, most ARF errors (84%) were easy to fix, almost all errors that took more than a few hours to fix were detected after the first 90% of the development cycle, and the documentation was significantly helpful in fixing errors.

Table 7 — Methods of Error Correction
(Reported for 116 of 143 Errors)

Category	Number of Errors	Percent of Total Errors
1. Inspection of code	32	23
2. Study of the algorithm in the documentation	14	10
3. Running test cases	6	4
4. Other	9	6
5. 1. & 2.	16	11
6. 1. & 3.	2	1
7. 1. & 4.	1	1
8. 1. & 2. & 3.	29	20
9. 1. & 2.& 3.& 4.	7	5

Limitations and Problems

Although it is possible to design repeatable experiments using error analysis as a tool to evaluate hypotheses, no attempt was made to do so for the ARF. The expense and difficulty of performing such experiments is a significant limitation on the use of error analysis (or any other technique involving complex, intensive, human behavior). Our purpose here is not to describe the design of such experiments, but rather to discuss some limitations and problems of error analysis in its role as an aid to project managers, system designers, programmers, and others. We will also discuss some particular problems encountered with the data collection and analysis procedures used for the ARF. Because the ARF error analysis project was not designed as a formal, repeatable experiment, it is hard to compare the results with similar efforts such as Endres [10]. The background of the people involved, the hardware and software environment used, the application being programmed, and many other factors confound attempts to draw such comparisons. As noted by Jones [16], even simple comparisons using parameters such as programmer productivity or error rates must be suspect because of the different conditions surrounding different projects. We view these difficulties as the principal limitation of the error analysis technique described here.

One problem for which there is no simple solution is the Hawthorne (or observer) effect [17]. When project personnel become aware that an aspect of their behavior is being monitored, their behavior will change. If error monitoring is a continuous, long-term activity that is part of the normal scheme of software development and not associated with evaluation of programmer performance, this effect may become insignificant. We believe this was the case with the ARF project. A second significant problem is accuracy of the data. Those who fill out error-report forms, particularly when they are new to the process, tend to interpret the questions and categories on the form differently than the designer of the form. (We assume that the error analyst has some standard set of criteria for categorizing errors.) As a result, all error reports must be carefully reviewed by an error analyst shortly after they have been completed. Questions or doubts about the information on the form should be resolved in an interview with the error reporter. The interview must take place soon enough after the error is observed and corrected for the error reporter to remember all details of the error. This situation can be complicated if the error committer, error corrector, and error reporter are different people. A further problem involved with completion of forms is the amount of overhead involved. Project managers with tight schedules resent activities that distract the attention of project personnel from the task at hand. One reason the ARF data are reasonably complete and accurate is that the project management strongly supported error data collection and analysis. Another problem is convincing programmers and designers that the error data will not be used against them. This was not a significant problem on the ARF project for several reasons, including management support, processing of the error data by a person independent of the ARF, identifying error reports in the analysis process by number rather than name, informing newly hired project personnél that completion of error reports was considered part of their job, and high project morale. Indeed, programmers often delighted in verbally describing to the error analyst all the gory details of the search for some particularly obscure error. Furthermore, project management did not need error data to evaluate performance. Indeed, the project manager was able to guess which programmers committed the most errors without seeing any error reports, based on her own estimate of their capabilities.

As in any project involving analysis of data, there was information that was never asked for, but that would have been useful. As an example, the length of time an error was in the system could not be calculated because document version numbers and release dates for code and documentation were not requested and were nowhere consistently maintained. The origin of some errors was difficult to establish because information concerning the first document or piece of code in which the error appeared was not requested. We were unable to analyze issues involving program modification because information about modifications for purposes other than error correction was not recorded anywhere. These examples underscore the need for careful planning of data collection before starting to gather data.

Most data concerning issues where little judgment is required to classify the error were obtained easily and accurately. The techniques used to find and correct errors and the amount of time (quantized into three categories) required to correct errors fall in this category. Most analyses concerning the mechanics of error detection and correction could be accomplished merely by counting check marks on error report forms. Issues requiring greater judgement, such as the sources of misunderstandings, sometimes required understanding by the error analyst of the events surrounding the error as well as the reasons for the error. It was consideration of these cases when the error-data-collection process was being designed that prompted the requirement that error data be collected in conjunction with system development.

In general, all error analyses planned at the beginning of the project were completed. Other analyses that were discovered to be of interest, such as those described in the preceding, could not be carried out because the appropriate data had not been collected and could no longer be obtained, i.e. unplanned, ad hoc data analysis was generally fruitless.

The basic limitations of error analysis as it was used for the ARF project appear to be the following:

- Unless a repeatable, formal experiment is carefully planned, one cannot expect to perform statistical hypothesis testing or to easily compare results between projects

- Data analyses must be carefully planned at the beginning of the project; lost data cannot be recaptured

- Error reports must be processed shortly after they are completed so that questions and ambiguities in classification may be resolved while the memory of the error is fresh

- The overhead involved in data collection may perturb very tight project schedules.

CONCLUSIONS

We have described here how historical error analysis may be used to evaluate software development methodologies. The key points of the technique are to design the data collection scheme prior to collecting data and to collect data in parallel with system development. Several previous studies, such as the RADC study [8], have used only ad hoc data, resulting in situations with large numbers of categories and few errors in each category.

Although it does not provide a true experimental evaluation, the technique used for the ARF project does yield insight into the effects of software development methodologies. The results of analyzing project errors are useful to project management, designers, programmers, software engineers, and others. The results may be used as a measure of the effectiveness of various development methodologies in preventing and detecting software errors, and as a measure of achievement of some types of project goals.

The ARF error results support the view that the ARF developers met their design objectives with respect to modularization and run-time error detection. Most errors took less than a few hours to fix. Intermodule interfaces apparently caused few problems, and errors and error corrections rarely spanned more than one module. More than half of the table access errors were detected by the run-time error-detection mechanism. The worth, with respect to error prevention and cost of error correction of the specifications and other documentation and the quality control rules, cannot be easily estimated. The documentation did seem to be significantly helpful in correcting errors.

The difficulties involved in conducting large-scale, controlled, software engineering experiments have as yet prevented evaluations of software development methodologies in the environments where they are often claimed to work best. As a result, software engineers must depend on less formal techniques that can be used in real working environments to establish long-term trends. Examples of such techniques [10, 15, 18] already exist in the literature. We view error analysis as one such technique and feel that more techniques, and many more results obtained by applying such techniques, are needed.

ACKNOWLEDGMENTS

I would like to thank all participants in the ARF project for their cooperation and patience in filling out error report forms. Alan Parker particularly was of great help in describing ARF errors and their causes, and in reviewing the categorizations of all the ARF errors. Drs. V. Basili, D. Parnas, and J. Shore, and Mr. L. Chmura contributed many helpful suggestions on an early draft of this report.

REFERENCES

1. C. G. Bell and A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, New York, 1971.

2. H. Elovitz, "The Architecture Research Facility: An Experiment in Software Engineering," unpublished paper, NRL.

3. D. L. Parnas, "On the Design and Development of Program Families," *IEEE Trans.* **SE-2** (No. 1), 1-9 (Mar.1976).

4. ———, "On the Criteria To Be Used in Decomposing Systems into Modules", *Commun. ACM* **15**, (No. 12), 1053-1058, (Dec. 1972).

5. J. McHugh, "FORTRAN Coding Standards for the ARF," NRL Technical Memorandum 5403-457:JMcH:dmf, December 9, 1975.

6. V. Basili and A. Turner, "SIMPL-T, A Structured Programming Language," University of Maryland Computer Note CN-14, Jan. 1974.

7. *BLISS-10 Programmer's Reference Manual*, Digital Equipment Corporation, Maynard, Mass., 1974.

8. G. Craig, W. Hetrick, M. Lipow, T. Thayer, et al., "Software Reliability Study," Rome Air Development Center Technical Report RADC-TR-74-250, Oct. 1974.

9. S. L. Gerhart and L. Yelowitz, "Observations of Fallibility in Applications of Modern Programming Methodologies," *IEEE Trans.* **SE-2** (No. 3), 195-207 (Sept. 1976).

10. A. Endres, "Analysis and Causes of Errors in System Programs," *Proc. Intern. Conf. Reliable Software*, pp. 327-336, Apr. 1975.

11. W. Amory and J. Clapp, *A Software Error Classification Methodology,* MRT-2648 Vol VII, The MITRE Corp., Bedford, Mass., June 1973.

12. E. Youngs, "Error Proneness In Programming," Ph.D Thesis, University of North Carolina at Chapel Hill, 1970.

13. P. Wright and P. Barnard, "'Just fill in this form' - A review for designers," *Appl. Ergonomics* **6.4**, 213-220 (1975).

14. W. Howden, *Theoretical and Empirical Studies of Program Testing, Proc. 3rd Internatl. Conf. Software Engrg.*, pp. 305-310, May 1978. Long Beach, Calif. IEEE Computer Society.

15. C. Walston and C. Felix, "A Method of Programming Measurement and Estimation," *IBM Syst. J.* **16** (No. 1), 54-73 (1977).

16. C. Jones, "*Productivity Measurements,*" in *Proc. GUIDE 44*, San Francisco, May 1977.

17. J. Brown, *The Social Psychology of Industry,* Penguin Books, Baltimore, Md., 1954.

18. V. Basili, M. Zelkowitz, F. McGarry, et al., *The Software Engineering Laboratory,* Univer-sity of Maryland Technical Report TR-535, May 1977.

Appendix
MODULES OF THE ARF

The main principle used in the modularization of the ARF was information hiding. There were eight modules in the original design, which are described in the following paragraphs. See Ref. 2 for a more complete description.

1. The ISP compiler translates ISP source code into low-level, simulatable, target machine descriptions. The compiler existed before ARF development started, and few extensions were made to it.

2. The ISP compiler postprocessor provides semantic checking and error diagnostic facilities not included in the compiler, but deemed desirable for ARF users.

3. The reformatter program translates the output of the compiler into a set of tables that can be used to simulate the target machine.

4. The simulator interprets the reformatted compiler output to simulate the target machine. Programs to be run on the simulated machine are stored on mass storage files and referenced as needed. The semantics of target machine instruction simulation are contained solely in the simulator.

5. The command-language interpreter provides the interface to the user. All communications with the user and interpretation of user requests are concealed in this module.

6. The table access module manages the usage of the tables that describe the components and the state of the machine being simulated. The representation of these tables is hidden in this module. All access to the tables is provided by a set of table interface functions, which belong to the table access module.

7. The paging subsystem provides a mechanism for representing a large target machine memory when limited memory is available for ARF use. The pager includes facilities for paging, loading, and saving the tables managed by the table access module. The paging subsystem was expected to be particularly valuable in implementing the PDP-11 version of the ARF. All memory mapping of the tables is handled by the paging module.

8. A PDP-10 to PDP-11 communications package was designed to reduce the effort in producing a PDP-11 version of the ARF. The ISP compiler, postprocessor, and the reformatter were all designed as PDP-10 programs that would not be moved to the PDP-11. The files produced as a result of executing those programs would be transmitted over a data link to the PDP-11 by the communications package. The communications interface between the two machines would be known only to the communications package. Since a PDP-11 version of the ARF was not attempted, the communications module was never implemented.